Using catamorphisms, subtypes and monad transformers for writing modular functional interpreters.

Luc Duponcheel
Utrecht University
Padualaan 14
3584 CH Utrecht
The Netherlands
`luc@cs.ruu.nl`
`http://www.cs.ruu.nl/people/luc`

November 16, 1995

Abstract When writing functional interpreters, there are two things which need to be taken care of when having modularity in mind: syntax and semantics. Previous papers have presented methods, based on subtypes and monad transformers, for dealing with semantics. This paper combines these methods with a method, based on catamorphisms, for dealing with syntax. Subtypes, monad transformers and catamorphisms make it possible to write library code which can be used for building functional interpreters which are composed of reusable components.

1 Introduction

In [11] Moggi proposes to model the semantics of a program as a computation. In [12] Wadler shows how to write programming language interpreters as functions `"interpret :: syntax -> compute value"`. The type `syntax` represents the syntax of the interpreted programming language. The unary type constructor `compute` turns a type `value` of semantic values into a type of computations which yield such values. Many authors have proposed methods for writing such functional interpreters in a modular way. Among them are, Jones and Duponcheel [3], Steele [10], and Liang, Hudak and Jones [8]. The interpreter of [10] is written in `Haskell` (see [5]), a lazy functional programming language which supports type classes. The interpreters of [3] and [8] are written in `Gofer` (see [4]), an `Haskell` dialect which supports constructor classes, a generalisation of type classes.

Modular functional interpreters are build by combining reusable components. Every component interprets a specific part of the programming language. Modularity is realised by partitioning the set of syntactic constructs of the programming language into subsets of related constructs. Typical subsets describe arithmetic expressions, variable definitions or function application and abstraction. If we write an interpreter component then we do not know the value type which is used by a final interpreter which makes use of the component. Similarly, we do not know the computational features which are used by the computation of a final interpreter which makes use of the component. Interpreter components should be written in such a way that they impose constraints upon their value type and computation rather than use fixed ones. Here is an example: if we write a component for interpreting arithmetic expressions, then the value type has to be a super-type of `Int`. If the arithmetic expressions handle division, then the ongoing computation must be able to throw an error message when a number is divided by 0. In [8] the authors make use of a `SubType` class to handle constraints upon the value type in a modular way. Similarly, they make use of a `MonadTransformer` class to handle constraints upon the computation in a modular way. Monad transformers generalise the monad composition methods which the authors of [3] and [10] make use of. In [11] Moggi already suggested to make use of monad transformers to deal with the problem of modular monadic semantics.

The interpreter in [8] is not composed of reusable components. It uses one monolithic data definition for representing the syntax of the interpreted language with. The main contribution of this paper is the following: we make use of a `Functor` and an `Algebra` class to deal with the syntax of the interpreted language in a modular way. The syntax of a programming language is modelled as the syntactic algebra of a functor. The semantics of the language is modelled as a semantic algebra of the functor : with every syntactic construct of the language corresponds a semantic action upon the algebra. This algebraic approach allows syntactic features to be combined. Moreover this algebraic approach can be combined with the monadic approach in a natural way. Functions which are defined by making use of functors and algebras are sometimes referred to as catamorphisms. In [9] Malcolm shows the usefulness of catamorphisms for functional programming. In [1] Meijer et. al. brought Malcolms work to the attention of the mainstream functional programming community. In [2] Meijer and Hutton present the ideas of [1] in an accessible way.

This paper is organised as follows: in section 2 we show how to use catamorphisms for handling syntax in a modular way. In section 3 we briefly show how to handle semantics. Section 4 presents some components of a modular λ-calculus interpreter which is similar to the one presented in [8]. The components make use of a combination of our classes for dealing with syntax and the classes of [8] for dealing with semantics. The complete code can be ftp'd from my home page. Finally, in section 5 we draw some conclusions.

2 Dealing with Syntax

In this section we show how to interpret arithmetic expressions in a modular way. Starting with a monolithic interpreter (in section 2.1), we finish with a modular one (in section 2.3).

2.1 A first try

Suppose that we have written the following code for a simple expression evaluator.

```
> data Expr1
>   = Num Int
>   | Expr1 `Add` Expr1

> num = id
> x `add` y = x + y :: Int

> eval1 :: Expr1 -> Int
> eval1 (Num n) = num n
> eval1 (e `Add` f) = eval1 e `add` eval1 f
```

The evaluator replaces the syntactic constructs `Num` and `Add` of `Expr1` by semantic actions `num` and `add` on `Int`. We can easily change this expression evaluator such that it can e.g. handle division as well.

```
> data Expr
>   = Num Int
>   | Expr `Add` Expr
>   | Expr `Dvd` Expr

> x `dvd` y
>   = if y == 0
>       then error "divide by 0"
>       else x/y

> eval :: Expr -> Int
> eval (Num n) = num n
> eval (e `Add` f) = eval e `add` eval f
> eval (e `Dvd` f) = eval e `dvd` eval f
```

The evaluator now also replaces the syntactic construct `Dvd` by a semantic action `dvd`. Note, by the way, that we have used the build-in `error` function of `Gofer` to throw an error message when a number is divided by 0. This way of throwing error messages is a non-acceptable computational behaviour. Interpreters are interactive programs and we do not want to finish their main read-eval-print loop when an exception such as dividing a number by 0 occurs.

The method for adding new features presented above, albeit simple and systematic, has a serious disadvantage: we have not reused any part of the code of the foregoing evaluator at all. It does not make much sense to define

```
> data Expr2 = Expr2 `Dvd` Expr2

> eval2 :: Expr2 -> Int
> eval2 (e `Dvd` f) = eval2 e `dvd` eval2 f
```

and try to combine `eval1` and `eval2` in one way or another. The problem comes from the fact that the components which we try to combine are of a recursive nature. Combining recursive data and functions does not result in the recursive data and functions we are looking for.

2.2 A second try

Fortunately, there exists an easy way out of the problem of subsection 2.1: instead of first defining recursive components and afterwards trying to combine them, we can also proceed the other way round by first combining non-recursive components and afterwards introducing recursion.

2.2.1 Data definitions

Dealing with the data definitions `Expr1` and `Expr2` of section 2.1 goes as follows: we replace recursion by an extra type parameter.

```
> data E1 x
>   = Num Int
>   | x `Add` x

> data E2 x = x `Dvd` x
```

Recursion can now be introduced as follows:

```
> data Expr1 = In1 (E1 Expr1)
> data Expr2 = In2 (E2 Expr2)
```

The data constructors `In1` and `In2` are needed to avoid recursive types. A simple calculation shows that, up to `In2`, the types `Expr2` and ``Expr2 `Dvd` Expr2`` are the same.

```
  Expr2 = In2 (E2 Expr2)
        = In2 (Expr2 `Dvd` Expr2)
```

2.2.2 Functions

Dealing with the functions `eval1` and `eval2` of section 2.1 is a little bit more complex. First we define functions `map1` and `map2` in which we replace recursion by an extra function parameter.

```
> map1 :: (x -> y) -> (E1 x -> E1 y)
> map1 g (Num n) = Num n
> map1 g (e `Add` f) = g e `Add` g f

> map2 :: (x -> y) -> (E2 x -> E2 y)
> map2 g (e `Dvd` f) = g e `Dvd` g f
```

Note that we have not replaced the syntactic constructs `Num`, `Add` and `Dvd` by their semantic counterparts `num`, `add` and `dvd`. This is done in the functions `phi1` and `phi2`.

```
> phi1 :: E1 Int -> Int
> phi1 (Num n) = num n
> phi1 (e `Add` f) = e `add` f

> phi2 :: E2 Int -> Int
> phi2 (e `Dvd` f) = e `dvd` f
```

The functions `num`, `add` and `dvd` can be seen as semantic actions on `Int`. The functions `phiE1` and `phiE2` apply these actions. Recursion can now be introduced as follows:

```
> eval1 :: Expr1 -> Int
> eval1 (In1 e1) = phi1 (map1 eval1 e1)

> eval2 :: Expr2 -> Int
> eval2 (In2 e2) = phi2 (map2 eval2 e2)
```

We invite the reader to have a closer look at all the types which are involved in the definition of the `eval` functions. The `Gofer` type checker can infer the type of the `eval` functions from their definition. The bodies of the `eval` functions have to be read as follows: once the `In` data constructor is stripped of, the actual structure of an expression is visible. The function `eval` can now do its job recursively for all the subexpressions via `map`. Finally, the numbers which are obtained by recursively applying `eval` to those subexpressions are assembled via `phi`.

Up to now it looks as if we have only complicated matters. We finished up with a function `eval1` which is similar (but more complex) than the one in subsection 2.1 and with a function `eval2` which is as useless as the one in subsection 2.1.

But here comes the reward for all our work: instead of concentrating on the data definitions `Expr1` and `Expr2` and the functions `eval1` and `eval2` we should concentrate on the data definitions `E1` and `E2` and the functions `phi1` and `phi2`. They *can* be combined resulting in an expression evaluator which can handle numbers, addition and division. We will show how this can be done in the following two subsections. Before we do so we present a general way of combining data definitions and functions:

```
> data Sum x y = L x | R y

> (<+>) ::   (x -> z) -> (y -> z)
>         -> (Sum x y -> z)
> l <+> r
>   = \s -> case s of
>             L x -> l x
>             R y -> r y
```

2.2.3 Combining data

First we define a combined data definition `E` (making use of `Sum`). The recursive data definition `Expr` can then be defined in the same way as `Expr1` and `Expr2`.

```
> type E x
>   = Sum (E1 x) (E2 x) in mapE, phiE

> data Expr = InE (E Expr)
```

2.2.4 Combining functions

Second we define combined functions `mapE` and `phiE` (making use of `<+>`). The recursive function `evalE` can then be defined in the same way as `eval1` and `eval2`.

```
> mapE :: (x -> y) -> (E x -> E y)
> mapE g = L . map1 g <+> R . map2 g

> phiE :: E Int -> Int
> phiE = phi1 <+> phi2

> evalE :: Expr -> Int
> evalE (InE e) = phiE (mapE evalE e)
```

All this looks very promising: when writing the code of interpreter components we have to think in terms of semantic actions (for functions like `phi1` and `phi2`) which correspond to syntactic constructs (of data definitions like `E1` and `E2`). The code of a final interpreter can be obtained by making use of `Sum` (for the data definitions) and `<+>` (for the semantic actions). The code which handles recursion is of a general nature and has no relationship whatsoever with the specific interpreter we have in mind.

2.3 A third try

Having a closer look at the code of section 2.2 reveals that there is a lot of room for generalisation.

1. The `Expr` data definitions are all similar.
2. The types of the functions `map` and `phi` and the bodies of the functions `eval` are all similar.
3. The way in which we combined the functions `map` and `phi` does not depend upon them at all.

These issues are addresses in this section.

2.3.1 Recursive data definitions

The data definitions `Expr1` and `Expr2` are all similar. This similarity can be captured by the following general recursive data definition:

```
> data Rec f = In (f (Rec f))

> type Expr1 = Rec E1
> type Expr2 = Rec E2
> type Expr = Rec E
```

A simple calculation shows that, up to `In`, the types `Expr2` and `E2 Expr2` are the same.

```
Expr2 = Rec E2
      = In (E2 (Rec E2))
      = In (E2 Expr2)
```

Note that exactly the same calculation can now be done for `Expr1` and `Expr`.

2.3.2 Recursive functions

The types of the functions `map1` and `map2` are all similar. It would be a pity not to use this fact in one way or another. This similarity can be captured by an appropriate constructor class. Constructor classes are one of the most powerful features of `Gofer`. They are a generalisation of the type classes of `Haskell`. Here is a typical example of a type class:

```
> class Eq x where
>   (==) :: x -> x -> Bool
```

A type `x` is an instance of the `Eq` class if an appropriate equality operator is defined on it. Here are some typical instances:

```
> instance Eq Int where
>   (==) = primIntEqOp

> instance (Eq x, Eq y) => Eq (x,y) where
>   (x,y) == (u,v) = x == u && y == v
```

The first instance makes use of a primitive equality operator on numbers. The second instance shows how to build complex instances out of simpler ones. The advantage of defining the `Eq` class comes from the fact that we can overload the `(==)` operator when defining general purpose functions which work on types on which an equality operator is defined. Here is an example:

```
> elem :: Eq x => x -> [x] -> Bool
> x `elem` [] = False
> x `elem` (y:ys) = x == y || x `elem` ys
```

If we were not allowed to use type classes, then we had to pass the `(==)` operator as an extra argument to the function `elem`. This clutters up the code of `elem` (not to mention the functions which makes use of it). The `Haskell` implementation passes along these extra arguments under water in one way or another. This feature is extremely useful (especially when several type classes are involved). Moreover, the `Haskell` type checker can infer the type of `elem` from its definition. The qualifier `Eq x` reflects the fact that we used `(==)` in the definition of `elem`. It cannot be stressed enough how useful this feature is when writing functional programs.

Constructor classes are similar to type classes. The parameters of the class are constructors (rather than types). Constructor classes are supported by `Gofer`. Here comes the definition of a first constructor class and a typical instance.

```
  class Functor f where
    map :: (x -> y) -> (f x -> f y)

  instance Functor [] where
    map f = g
      where
       g [] = []
       g (x:xs) = f x : g xs
```

The `Functor` class allows us to make use of the `map` notation, which is normally used when working with lists, in other situations as well.

The types of the functions `map1`, `map2` and `mapE` of section 2.2 are all instances of the general type of `map`. Here are the corresponding class instances.

```
> instance Functor E1 where
>   map = map1
> instance Functor E2 where
>   map = map2
> instance Functor E where
>   map = mapE
```

The functions `map1`, `map2` and `mapE` follow the pattern of recursion defined by `E1`, `E2` and `E`. Thus, the member `map` of the `Functor` class can be seen as following the pattern of recursion defined by the functor `f`.

The types of the functions `phi1`, `phi2` and `phiE` of section 2.2 are all similar. We will model this by making use of a second constructor class and corresponding instances.

```
> class Functor f => Algebra f a where
>   phi :: f a -> a
```

The types of the functions `phi1`, `phi2` and `phiE` of section 2.2 are all instances of the general type of `phi`. Here are the corresponding class instances.

```
> instance Algebra E1 Int where
>   phi = phi1
> instance Algebra E2 Int where
>   phi = phi2
> instance Algebra E Int where
>   phi = phiE
```

The functions `phi1`, `phi2` and `phiE` apply semantic actions on `Int` corresponding to syntactic constructs of the functors `E1`, `E2` and `E`. Thus, the member `phi` of the `Algebra` class can be seen as applying semantic actions on the algebra `a` corresponding to syntactic constructs of a functor `f`.

Once we have defined the `Algebra` class we can define a general recursive function `eval` in terms of it as follows:

```
> eval :: Algebra f a => Rec f -> a
> eval (In e) = phi (map eval e)
```

This definition looks, at least in my opinion, very appealing. We do not have to pass along extra parameters `phi` and `map`. The `Gofer` type checker can infer the type of `eval` from its definition. The extra parameters are simply replaced by the class constraint `Algebra f a`. The body of `eval` abstracts on the common pattern of the bodies of the functions `eval1`, `eval2` and `evalE` of section 2.2.

The functions `eval` are sometimes called `fold` and referred to as *catamorphisms*. The name `fold` suggests that there also exists a similar `unfold` function. This is indeed the case. We strongly recommend the interested reader to have a look at the, so called, banana papers [1] and [2].

2.3.3 Combining functors and algebras

The function `mapE` (resp. `phiE`) of section 2.2 is defined in terms of `map1` and `map2` (resp. `phi1` and `phi2`) in a way which does not depend upon them at all. This observation leads to the following `Functor` and `Algebra` class instances:

```
> type SumC f g x = Sum (f x) (g x)
>   in mapC, phiC

> mapC :: (Functor f, Functor g)
>         =>   (x -> y)
>           -> (SumC f g x -> SumC f g y)
> mapC g = L . map g <+> R . map g
```

```
> phiC :: (Algebra f a, Algebra g a)
>        => SumC f g a -> a
> phiC = phi <+> phi

> instance (Functor f, Functor g)
>          => Functor (SumC f g) where
>   map = mapC

> instance (Algebra f a, Algebra g a)
>          => Algebra (SumC f g) a where
>   phi = phiC
```

Using these general instances we can avoid writing down similar definitions when combining functions like `map1` and `map2` (resp. `phi1` and `phi2`) over and over again.

It may look as if all problems related to modularity are solved now: we can look at the syntax of the interpreted language as being partitioned into several parts and write code of components which interpret these parts. All the rest is done automatically by making use of appropriate general purpose classes. We have oversimplified matters. We have not dealt with the semantic aspects of the components at all. The components for evaluating arithmetic expressions used `Int` as their semantic value type. What happens if we want to use such a component for an interpreter which has to evaluate boolean expressions as well? We should write components in such a way that they put constraints upon the value type (rather than use a fixed one). Similarly, the components for evaluating arithmetic expression use no computational features. Therefore we have not mentioned computations at all. What happens if we we want to use such a component for an interpreter which requires computational features, such as throwing an error when dividing a number by 0 in a way which does not stop the main read-eval-print loop? Again we should write components in such a way that they put constraints upon the computation (rather than use a fixed one). Type and constructor classes turn out to be excellent tools for specifying the constraints upon the value type and the computation with.

3 Dealing with Semantics

In this section we show how to deal the with semantics of the interpreted language. This work has already been presented by other authors, most notably by Liang, Hudak and Jones in [8]. Therefore we do not go into the details. Instead we present some of the reusable components of a λ-calculus interpreter which is similar to the one in [8]. As argued in section 2 these components are `Algebra` instances which represent the application of semantic actions. Following the ideas of Moggi we work with computations. As a consequence, we will define semantic actions on computations rather than on values. In this way we can combine our algebraic (semantic action based) approach with the monadic (computation based) approach in an elegant way.

3.1 Sub-typing

If we write an interpreter component, say for interpreting numbers, then we do not know the value type of a final interpreter which makes use of the component. We only know that, if we want to interpret numbers in a faithful way, this value type has to be a super-type of `Int`. Subtypes (and super-types) can be modelled using the following type class.

```
> class SubType sub sup where
>   inj :: sub -> sup
>   prj :: sup -> sub
```

3.2 Monads

The common behaviour of computations can be modelled using a `Monad` class. First, there must be some way to return a result from a computation. Second, there must be some way to bind the value which is returned by a computation to a continuation in order to yield a new computation.

```
  class Functor m => Monad m where
    result :: x -> m x
    bind   :: m x -> (x -> m y) -> m y
```

We do not go into the details of monads. An excellent introduction to the usage of monads for functional programming can be found in [12]. An excellent introduction to the usage of the `Monad` class can be found in [6] and [7]. Using monads gives functional programming an imperative flavour. `Gofer` supports a, so called, `do` notation which makes this imperative flavour more apparent. A typical piece of code which makes use of this `do` syntax looks like:

```
>  do
>    x <- mx
>    y <- my
>    result (f x y)
```

The results `x` and `y` returned from the computations `mx` and `my` are both accessible by the function `f`. This is realised using lambda abstraction and `bind`. In fact, the code above is syntactic sugar for:

```
>  mx `bind` \x ->
>  my `bind` \y ->
>  result (f x y)
```

For more details on the `do` notation we refer the interested reader to the release notes of `Gofer 2.30` [4]. Before presenting the code of some interpreter components we define a useful function which combines subtype and monad features.

```
> resultInj ::  (Monad m, SubType sub sup)
>               => sub -> m sup
> resultInj = result . inj
```

3.3 A component for numbers

In this subsection we present the code of a component for interpreting numbers. The component also handles division. Therefore it needs more features than the ones which are offered by the `Monad` class: we want to throw an error message when dividing a number by 0. Computations which can throw an error message can be modelled using a constructor class which is derived from the `Monad` class.

```
> class Monad m => ErrMonad m where
>   throw :: String -> m x
```

Now we have all the ingredients which are needed for writing a number interpreter component. The code of this component is very general (it turns out that we have to help the `Gofer` type checker a bit by introducing safe type casts).

```
> data N x
>   = Num Int
>   | x `Add` x
>   | x `Dvd` x

> num n = resultInj n

> mx `add` my
>   = do
>      x <- mx
>      y <- my
>      resultInj (prj x |+| prj y)
>     where
>      (|+|) = (+) :: Int -> Int -> Int

> mx `dvd` my
>   = do
>      x <- mx
>      y <- my
>      if (prj y == 0)
>       then throw ("divide by 0")
>       else resultInj (prj x |/| prj y)
>     where
>      (|/|) = (/) :: Int -> Int -> Int

> instance (ErrMonad m, SubType Int v)
>          => Algebra N (m v) where
>   phi (Num n) = num n
>   phi (mx `Add` my) = mx `add` my
>   phi (mx `Dvd` my) = mx `dvd` my
```

Once again we would like to emphasise the fact that the `Gofer` type checker can be used to infer the requirements upon the value type and the computation. An interpreter which makes use of this number component should have a value type which contains the integers and should use a computation which can throw error messages.

3.4 A component for functions

A more spectacular component is the one for function abstraction and function application. It is a well known fact that, in order to treat functions as first class citizens, the the value type has to be a super-type of its own function type. This property is called reflexivity. In our monadic framework we need the value type has to be a super-type of its own monadic function type. Monadic functions map computations to computations rather than values to values. Monadic reflexivity can be modelled using the following class (whose only purpose is to define a synonym for a special instance of an already existing class).

```
> class SubType (m v -> m v) v
>      => Reflexive m v
> instance SubType (m v -> m v) v
>      => Reflexive m v
```

This class is special in the sense that it puts a common requirement upon both the value type and the computation. Function application and abstraction can (for example) be implemented by making use of an environment which consists of a table which holds monadic values. The ongoing computation has to be able to read the current environment and to compute with a given new environment. This computational behaviour can be modelled using a class which is derived from the `Monad` class.

```
> class Monad m => EnvMonad env m where
>   read :: m env
>   with :: env -> m x -> m x
```

We are now ready for our function interpreter component. Application is handled by first evaluating the function and then applying it to its argument which is evaluated with the table one obtains after evaluating the function. Abstraction is handled by returning a function which uses a table which is updated in an appropriate way. Notice that, for call-by-value we update with the result of a computation while for call-by-name we update with a computation.

```
> type Table x = [(String,x)]

> updateT ::  Monad m
>          => (String,m v) -> Table (m v)
>            -> m (Table (m v))
> updateT (x,m) tab = result ((x,m):tab)

> withT :: EnvMonad (Table (m v)) m
>        => Table (m v) -> m v -> m v
> withT = with

> data F x = App x x
>          | LamN String x -- call-by-name
>          | LamV String x -- call-by-value

> app mf ma
>  = do
>     f <- mf
>     tab <- read
>     prj f (withT tab ma)

> lamN x mb
>  = do
>     tab <- read
>     resultInj
>     (\m ->
>      do
>       newtab <- updateT (x,m) tab
>       withT newtab mb)

> lamV x mb
>  = do
>     tab <- read
>     resultInj
>     (\m ->
>      do
>       v <- m
>       newtab <- updateT (x,result v) tab
>       withT newtab mb)
```

```
> instance (EnvMonad (Table (m v)) m
>          , Reflexive m v)
>         => Algebra F (m v) where
>  phi (App mf ma) = app mf ma
>  phi (LamN x mb) = lamN x mb
>  phi (LamV x mb) = lamV x mb
```

An interpreter which makes use of this component should be able to use a table as an environment and should be reflexive in the sense that its value type has to be a super-type of its own monadic function type.

4 An example

We have not dealt with the problem of realising class constraints at all. How can we produce a value type and a monad which realise the constraints imposed by all the components a final interpreter makes use of?

4.1 Realising subtype constraints

Subtype constraints can be realised using the `Sum` data type. We can build towers of types such as:

```
> type Value = Sum Int (Only Bool)
```

The types `Int` and `Bool` are both subtypes of the type `Value`. For purely technical reasons (avoiding overlapping `SubType` class instances) we also have to introduce a dummy type synonym `Only`:

```
> type Only u = u in only,ylno

> only :: u -> Only u
> only = id

> ylno :: Only u -> u
> ylno = id
```

4.2 Realising monad constraints

Monad constraints can be realised using monad transformers. They are modelled using the following class:

```
> class MonadT t where
>   lift :: m x -> t m x
```

The idea is to define, for every special monad class (for example the `ErrMonad` class), a monad transformer (say `ErrT`) which is such that any transformed monad

`ErrT m` is an instance of `ErrMonad`. Using the `lift` member of the `MonadT` class we can (hope to) lift the computational features of the original monad `m` to the transformed monad `ErrT m`. In this way we end up with a monad which has one extra computational feature (in this case the ability to throw error messages). Here is a typical chain of transformed monads:

```
type Compute = EnvT (Table (ErrT Id))
```

The transformers `EnvT` and `ErrT` are actually monad compositions: at the left with the reader monad and at the right with the error monad respectively (see [3] and [8] for more information). The resulting monad can both use `Table` as an environment and throw error messages. We used the identity monad as a base monad. If we want to build an interpreter which can handle nondeterministic computations, then we should use the list monad as a base monad. Using nondeterminism one can show the difference between call-by-name and call-by-need semantics. Here is a typical session with an interpreter which has all the features which are needed to show this difference.

```
? main
ModularInterpreter
author: Luc Duponcheel

$ (\V x -> (x+x) [1,2])
  after parse:
  (\V x -> (x+x) [1,2])
  after eval:
  [2,4]
$ (\N x -> (x+x) [1,2])
  after parse:
  (\N x -> (x+x) [1,2])
  after eval:
  [2,3,3,4]
```

Finally, note that it is not a good idea to use the list monad to transform other monads with. Composing with the list monad does not work well since the associativity law for monad composition does not hold (see [3] for a minimal counterexample).

5 Conclusion and future work

We have showed how to use catamorphisms for dealing with the syntax of an interpreted language in a modular way. This results in functional interpreters which are composed of reusable components. We modelled catamorphisms by making use of constructor classes and we have combined those classes with the classes which are used in previous work to structure the semantics of the interpreted language with. Finally we have written a reusable component library using which one can build λ-calculus interpreters in a flexible way. The `Gofer` type system has turned out to be extremely useful when writing the code of the components. Writing modular interpreters can be summarised as follows:

- Partition the syntax of the interpreted language into parts which contain related constructs. Model the syntax of the parts as functors. Write interpreter components as algebras of semantic actions corresponding to syntactic constructs. Model the syntax of the whole language as the sum of the functors of the parts. Model the semantics of the whole language as the sum of the algebras of the parts. Use a fixed point of the final functor and algebra to obtain a final interpreter.

- Use subtypes to abstract away from the actual value types of the interpreter components. Realise the subtype requirements of a final interpreter by building a semantic domain as a tower of types.

- Use appropriate computational abstractions when writing interpreter components. Realise the computational abstractions by building a monad using a chain of monad transformers.

The methods for writing modular interpreter code do not only work for concrete interpreters such as the one presented in this paper. They also work for abstract interpreters (such as type checkers). The author is currently working on modular type checker components. Moreover the type checker components avoid the overhead of substitutions by representing types as graphs. We believe that our framework also makes it possible to write a whole range of type checkers (such as type checkers for linear types) in a very flexible way.

Acknowledgements I would like to thank Graham Hutton for reading an early draft of the paper and for giving me many helpful suggestions. Furthermore I would like to thank Doaitse Swierstra for giving me the opportunity to work on this paper.

References

[1] Erik Meijer, Maarten Fokkinga and Ross Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In John Hughes, editor, *Proc. Conference on Functional Programming and Computer Architecture*. Springer-Verlag, June 1991. Lecture Notes in Computer Science 523.

[2] Erik Meijer and Graham Hutton. Bananas in Space: Extending fold and unfold to Exponential Types. *Proc. Conference on Functional Programming and Computer Architecture.* 1995.

[3] Mark P. Jones and Luc Duponcheel. Composing monads. Research Report YALEU/DCS/RR-1004, Yale University Department of Computer Science, New Haven, Connecticut, December 1993.

[4] Mark P. Jones. Official release notes gofer 2.30, ftp://ftp.cs.nott.ac.uk/nott-fp/languages/gofer/,

[5] Paul Hudak, Simon Peyton Jones, Philip Wadler. Report on the programming language Haskell: a non-strict, purely functional language, version 1.2. ACM SIGPLAN Notices, Vol. 27(5), May 1992.

[6] Mark P. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. *Proc. Conference on Functional Programming and Computer Architecture.* New York, June 1993. ACM Press.

[7] Mark P. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, January 1995.

[8] Sheng Liang, Paul Hudak, Mark Jones. Monad Transformers and Modular Interpreters. *Proc. 22nd ACM Symposium on Principles of Programming Languages.* San Francisco, January 1995.

[9] Grant Malcolm. Algebraic Data Types and Program Transformation, *Science of Computer Programming*, Vol 14, pp. 255–280, September, 1990.

[10] Guy Steele Jr. Building interpreters by composing monads. *Proc. 21st ACM Symposium on Principles of Programming Languages.* New York, January 1994.

[11] Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, University of Edinburgh, Scotland, 1990.

[12] Philip Wadler. The essence of functional programming. *Proc. 19th ACM Symposium on Principles of Programming Languages.* New Mexico, January 1992.